Efficient and Secure Search of Enterprise File Systems

Aameek Singh
College of Computing
Georgia Tech

Mudhakar Srivatsa
College of Computing
Georgia Tech

Ling Liu
College of Computing
Georgia Tech

Abstract

With fast paced growth of digital data, keyword based search has become a critical enterprise application. Research has shown that nearly 85% of enterprise data lies in flat filesystems [10] that allow multiple users with different access privileges. Any search tool for such systems needs to be efficient and yet cognizant of access control semantics imposed by the underlying filesystem. Current enterprise search techniques use two disjoint search *and* access-control *components by creating a single system-wide index and filtering search results for access control. This approach is ineffective as index and query statistics subtly leak private information. The other approach of using separate indices for each user is undesirable as it not only increases disk consumption due to shared files, but also increases overheads of updating indices whenever a file changes.*

We propose a distributed approach that couples search and access-control into a unified framework and provides secure multiuser search. Our scheme (logically) divides data into independent access-privileges based chunks, called access-control barrels (ACB). ACBs not only manage security but also improve overall efficiency as they can be indexed and searched in parallel by distributing them to multiple enterprise machines. We describe the architecture of ACBs based search and propose an optimization that ensures the scalability of our approach. We validate our design with a detailed evaluation using industry benchmarks and datasets. Our initial experiments show secure search with 38% improved indexing efficiency and low overheads for ACB processing.

1. Introduction

In recent years, the total amount of digital data has grown leaps and bounds, doubling almost every eighteen months [2]. As the cost of storage hardware drops, enterprises are storing more data and also keeping it for a longer period of time. It is for both business intelligence as well as regulatory compliance purposes. With this large amount of data available, keyword based search to quickly locate relevant data has become an essential IT capability.

According to estimates, as much as 85% of enterprise data is stored in unstructured repositories like enterprise and local filesystems [10]. These filesystems have multiple users with different privileges to data and access is controlled using native access control mechanisms like Unix/Windows permissions models. This access control needs to be enforced even while searching through the data. As a simple case in point, a user should not be able to search through data that is not accessible to that user. Also, there are additional subtle requirements that complicate this process and if unhandled, can result in information leaks. For example, looking at the results of a query a user should not be able to extract any information that could not have been inferred by that user by accessing the underlying filesystem. We refer to this principle as *Access Control Aware Search* or ACAS in short. Simply put, ACAS requires that no additional information can be extracted about the filesystem by using the search mechanism. More formally we define it as:

Definition: ***Access Control Aware Search (ACAS)***
Let $\mathcal{I}_F^U$ be the information that a user U can extract from filesystem F by accessing it directly (dictated by access rights for U) and let $\mathcal{I}_S^U$ be the information that U can extract by searching on indices over F over any period of time (based on the search mechanism). The access control aware search (ACAS) property requires that $\mathcal{I}_S^U \subseteq \mathcal{I}_F^U$.

Surprisingly most enterprise search products in the market, like Google Enterprise [4], Coveo [1], [8], do not satisfy the ACAS principle. These tools treat search and access-control as two disjoint components and can result in malicious users extracting unauthorized information using the search mechanism. In their approach, a single system wide index is created for all users and it is queried using traditional information retrieval (IR) techniques (the *search* component). Finally the results (the list of files containing query keywords) are filtered based on access privileges for the querying user (*access control*). However, the ordering and relevance score of results, based on Term-Frequency-Inverse-Document-Frequency (TFIDF) measures, reveal information that violates the ACAS property. Intuitively,

since the index was created based on the lexicon and documents of the complete system, simple post-processing of results would fail to adequately protect system-wide statistics against carefully crafted attacks. We describe this issue and demonstrate an example attack in §2.2.

A technique that satisfies ACAS can be found in common desktop search products like Google Desktop [3] and Yahoo Desktop [9]. These tools create distinct indices for each user on the system, with each user index including all files accessible to that user (the *access-control* component) and then querying only that index for the user (the *search* component). While this satisfies ACAS, it is highly inefficient as it requires every shared file to be indexed multiple times in the indices of each user that can access that file. Since in modern enterprises, a large amount of data is shared by many users, this approach not only causes greater disk consumption (due to increased index size), but the overheads of updating the indices when a file changes also become significant.

We propose a distributed enterprise search technique that couples search and access-control into a unified framework to provide secure and efficient search. We use a novel building block called access control barrel (ACB) that ensures access control aware search. An ACB is a set of files that have the same access privileges for users and groups in the system and by dividing filesystem data into independent ACBs, we can ensure that the index for a user is only derived from data accessible to that user in the underlying filesystem, satisfying the ACAS requirement.

The ACB-based approach is also space and update efficient as it ensures that each file is included in only a single index. This *minimality* property makes it especially suitable for shared multiuser environments. Further, by dividing data into independent barrels, data indexing can be distributed to multiple machines for parallel processing. This can significantly reduce total indexing time. We also describe an optimization technique that ensures the scalability of our approach even in complex enterprise environments.
In summary this paper makes the following contributions:

- **Access Control Aware Enterprise Search**: We have characterized the access control problem in enterprise search. We also propose a new Access Control Barrel (ACB) concept that prevents information leaks to unauthorized users.
- **Space and Update Efficiency**: By ensuring that majority of files are included in a single index, our approach provides superior space & update efficiency. Our proposed optimization also enhances scalability of our approach even in complex settings.
- **Distributed Enterprise Search Architecture**: In contrast to existing centralized approaches, we use a distributed architecture that parallelizes indexing and search for better performance and is better suited to enterprises with numerous underutilized machines.

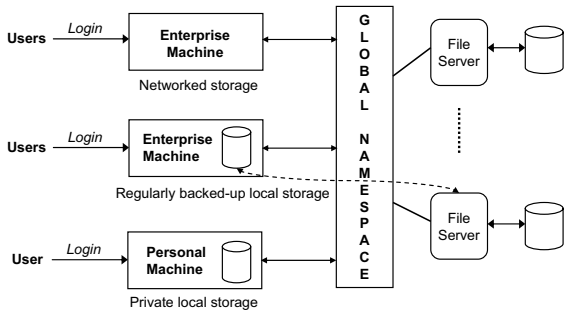

Figure 1: Enterprise Storage Architecture

2. Background and Related Work

In this section, we briefly describe modern enterprise architecture, that serves as the model environment assumed in this paper. Later, we will discuss related work and limitations of existing enterprise search techniques.

2.1. Modern Enterprise Architecture

Modern enterprises today are data-rich environments with storage capacities running into terabytes and petabytes. A typical enterprise has multiple file servers, most commonly accessed via *nix based Network File System (NFS) or Windows based Common Internet File System (CIFS) protocols. These file servers could be directly attached to storage (Network Attached Storage, NAS architecture as shown in Figure-1) or through a Storage Area Network (SAN). In the context of our work, we only require a common global namespace which is widely supported in both architectures.

On the client side, this namespace is accessed by many always-connected (and mostly wired) enterprise machines. These could be user workstations, shared laboratory machines or application servers. Most of these machines have networked storage (for example, NFS mounted). Some of these machines might also have some local storage that is regularly backed up to global file servers. We call such machines *enterprise machines* since they are under administrative control and can be potentially leveraged for indexing and search tasks. Other prominent client category consists of single-user personal machines like laptops. We call such clients *personal machines* and consider them to be unavailable for administrative usage.

2.1.1 Access Control

In an enterprise environment, not all data is accessible to all users. Access to data is controlled through the underlying filesystem's access control mechanisms. For example, NFS file servers follow the UNIX permissions

model and Windows-based file servers follow the Windows permissions model. In this work, because of its widespread deployment and easier availability for experimentation, we use the UNIX based NFS architecture (also supported by other *nix flavors like Linux, FreeBSD). We refer the reader to [20] for detailed UNIX permissions discussion. In context of indexing/search, we use the following notion of **searchability** consistent with the UNIX `find/slocate` [5] paradigms:

Definition: ***Searchability*** *– A file, F is* ***searchable*** *by user u_i (or group g_m) if there exist* read *and* execute *permissions on the path leading to F and* read *permissions on F.*

2.2. Limitations of Existing Approaches

In this section, we explore related keyword search solutions and analyze their pros and cons. Most desktop search products like Google Desktop [3], Yahoo Desktop [9] integrate access control during indexing. Each user has a separate index for accessible files with duplicates for files shared with other users. This ensures that each user has an index created from data that was accessible to that user, thus satisfying the ACAS requirement. Also, there are no additional costs at query runtime for access control based filtering. However, this causes *additional* disk consumption of $\sum(n_i - 1) * I_i$ where n_i is the number of users accessing file F_i and I_i is size of index for F_i. Additionally, each update to F_i causes updates to n_i indices. In a typical enterprise, where n_i is large, such costs can be prohibitive. A detailed analysis is presented in [22].

The enterprise search products like Google Enterprise [4], Coveo Enterprise [1] and others [8] integrate access control at query runtime by creating a single system-wide index and filtering results based on access privileges of the querying user. This provides maximum space and update efficiency. However, querying is more expensive, as access permissions for files in the results are obtained at runtime, which requires disk I/O for inode lookups. Also importantly, these products do not satisfy the ACAS requirement and by carefully crafting queries, a user can obtain information about the underlying filesystem which could not have been inferred otherwise. Below, we describe an example attack that can determine the total number of files containing a particular keyword even when the attacker does not have access to all files containing that keyword. For example, an attacker could monitor the enterprise filesystem to see the number of files containing the word "*bankruptcy*". A sudden increase in the number of such files could alert him/her to sell off company stock. This violates ACAS, as this information could not have been determined by the attacker through the underlying filesystem directly.

We assume that the relevance score of a result file f_i is computed by the standard TFIDF measure

$$rel(f_i) = \sum_{t_j \in Q} w_{ij} * w_{Qj} \qquad (1)$$

where t_j are the terms in the query Q and w_{ij} is the normalized weight of term t_j in f_i given by

$$w_{ij} = \frac{o_{ij} * log\left(\frac{|N|}{n_{t_j}}\right)}{\sqrt{\sum_{t_k \in f_i}(o_{ik})^2 * \left(log\left(\frac{|N|}{n_{t_k}}\right)\right)^2}} \qquad (2)$$

where o_{ij} is the number of occurrences of term t_j in f_i, $|N|$ is the total number of files in the system and n_{t_j} is the number of files that contain t_j. w_{Qj} is defined similarly.

The attacker, Alice, wishes to know the number of documents that contain the term t_q (e.g. "*bankruptcy*"). The attack works in three steps. First, Alice picks two *unique* terms t_1, t_2 (no file contains these terms) and creates two new files: f_1 containing terms $\{t_1, t_2\}$ and f_2 containing terms $\{t_2, t_q\}$. Note that after creating the files, o_{11}=o_{12}=o_{22}=o_{2q}=1, n_{t_1}=1 and n_{t_2}=2. In the second step, she queries for term t_1 and from (1) and (2) she can calculate $|N|$

$$|N| = 2^{\frac{1}{1-\sqrt{\frac{1}{rel(f_1)^2}-1}}} \qquad (3)$$

In the final step, Alice queries for term t_q and calculates n_{t_q} from (1), (2), (3). This completes the attack.

$$n_{t_q} = 2^{\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}} * |N|^{\left(1-\frac{1}{\sqrt{\frac{1}{rel(f_2)^2}-1}}\right)}$$

Such attacks are possible on most TFIDF based measures including the popular Okapi BM25 [21]. Additionally, even when relevance scores are not returned in the result, good approximations to n_{t_q} can be obtained by exploiting ordering of the results [12]. A recent effort [12] describes an ACAS compliant approach using a complex query transformation at runtime. Additionally, it maintains access control lists for all files in the filesystem in-memory which is extremely inefficient for large enterprise environments.

In contrast to these centralized approaches, we use a distributed architecture using a novel access control barrel primitive. Our technique leverages existing work in distributed information retrieval. [16] compares approaches of distributed retrieval and concluded that distributed IR systems can be as fast and effective (quality-wise) as monolithic systems. [23] suggests that effectiveness of distributed IR systems can drop by up to 30% when the number of collections exceeds 100. However, in our approach the number of collections (number of barrels per user) is on an average about 5–7 (§5) and thus we expect to obtain good quality results.

We would like to distinguish our work from earlier privacy preserving indexing work. Bawa et al [11] present techniques for constructing a privacy preserving index on

documents in a multi-organizational setting. Their goal is to construct a centralized index that can be made public without revealing private information. They apply access control at query runtime and incur higher overheads. Our approach focuses on integrating access control with search in a single enterprise setting and is more efficient.

3. Distributed Enterprise Search

Desktop search products are secure but inefficient for enterprise search, whereas enterprise search products are insecure in terms of ACAS. Bearing these issues in mind, we describe our distributed approach to enterprise search which provides both security and efficiency.

3.1. Design Overview

The main design principle of our approach is to efficiently integrate access control into the indexing phase such that the indices used to respond to a user's query are derived only from the data accessible to that user, satisfying the ACAS requirement. We accomplish this goal with a pre-processing step that (a) constructs a user access hierarchy for users and user groups in the system (§3.1.2) and (b) logically divides data into access-privileges based *access control barrels (ACB)*. We first provide a description of ACBs and then describe how they work in conjunction with the user access hierarchy.

3.1.1 Access Control Barrels

An ACB is a set of files that share common searchability access privileges (as defined in §2.1.1). That is, all files contained within an access control barrel can be accessed (and thus searched) by the same set of users and user groups. For example, one barrel could contain files accessible to user *bob* and another for a user group *students*. Intuitively, the idea of barrels is that if we can efficiently create collections of files based on their access privileges, to provide secure search to a user, we can pick the collections that this user has access to and serve the query using only those indices.

This might sound similar to the index-per-user (IPU) desktop search approaches [3, 9, 6] where all files accessible to a user are grouped into a single collection and files accessible to multiple users are duplicated in their collections. ACBs avoid their inefficiencies by following an additional neat property of *minimality*. This property ensures that each file is uniquely mapped to a single barrel, avoiding duplication (we defer the discussion of implementing such minimal ACBs to later). Now, files accessible to multiple users are grouped into *shared* collections and search for a user combines the user's private collections with these shared collections using distributed IR [16]. This is efficiently accomplished using the user access hierarchy which is described next.

3.1.2 User Access Hierarchy

The user access hierarchy data structure has two main tasks: (1) provide a mechanism to map files to ACBs, and (2) provide techniques to efficiently determine all barrels that contain files searchable by a user. In what follows, we first give a high level description of the data structure and later describe its construction for *nix permissions model.

For most access control models a user is associated with two types of credentials: (i) a unique user identifier (uid), and (ii) one or more group identifiers (gid) corresponding to the user's group memberships. We represent the set of all such user and group credentials as a directed acyclic graph called Access Credentials Graph or ACG in short. For example, there could be a node for credential uid_{bob} or $gid_{students}$. Every node V_i in this graph is associated with a corresponding barrel ACB_i. Now, mapping files to barrels is equivalent to assigning files to a node in ACG (the first task mentioned above).

For a node, V_u (associated with the uid credential of a user u), let V_u^* denote the set of all nodes in the *directed* graph ACG that are reachable from the vertex V_u. Our construction of ACG will ensure that a file F is searchable for a user u if and only if F is assigned to some vertex $v \in V_u^*$. With this property, results for u's query can be computed by combining indices from barrels associated with nodes in V_u^*. The set V_u^* can be easily determined using a depth first search on ACG. This accomplishes the second task of this data structure.

Next, we explain the process of constructing the graph ACG with aforementioned properties from *nix-like user credentials. In a *nix-like access control model a credential C can be expressed in Backus Naur Form (BNF) as:

$$\begin{aligned} C &= root \mid all \mid P \\ P &= uid \mid gid \mid P \wedge P \mid P \vee P \qquad (I) \end{aligned}$$

Note that $root$ is a special user with super-user privileges and all indicates a credential for all users and groups. We need the $\vee$ operator on the principles to handle POSIX Access Control Lists [13] that allow associating multiple users and groups with a file F. We need the $\wedge$ operator on the principles to handle the implicit conjunction operation that occurs while traversing the directory hierarchy leading to file F (for example, directory `X/Y` where `X` has access only for user group *students*, and `Y` has access only for user group *grad-students*; only users that belong to both groups can access data under `Y`). We define an implication operator $\Rightarrow$ which specifies if one credential can *dominate* another. For example, $\forall u, root \Rightarrow u$ says that $root$ can access data that any other user can.

Permissions on a file can also be expressed based on a credential defined as above. For example, for a file F that allows access to users x, y and group z, we say that it has a credential C_F=$\{uid_x \vee uid_y \vee gid_z\}$ and is interpreted to

say that access is allowed to users who have a credential that dominates this file's credential (user x has access to F since $uid_x \Rightarrow C_F$). Now, if we can create a barrel for each such file credential in the system, we can uniquely map a file to a barrel, achieving ACB minimality. While theoretically the total number of barrels (one for each possible access control setting, thus exponential in number of users and groups) can be very large, practically, this is hardly the case as many files in the system share common file credentials. Our tests put this number at 5–7 and similar observations were made in [15]. Regardless, in §3.2, we will describe an optimization techniques that address this potential scalability issue.

Finally, we construct the graph ACG as follows. First, the set of vertices and edges are initialized by adding vertices for all user and group credentials and adding edges for the simple $\Rightarrow$ relationships: $root \Rightarrow u\ \forall u \in U$; $u \Rightarrow g\ \forall g \in G(u)$; for any group g, $g \Rightarrow all$, where $G(u)$ denotes the set of all groups to which the user u belongs. Formally,

$$\begin{aligned} V_{ACG} &= \{V_{root}, V_{all}\} \cup \{V_u \mid \forall u \in U\} \cup \{V_g \mid \forall g \in G\} \\ E_{ACG} &= \{V_{root} \rightarrow V_u \mid \forall u \in U\} \cup \{V_u \rightarrow V_g \mid \forall u \in U, \\ &\quad \forall g \in G(u)\} \cup \{V_g \rightarrow V_{all} \mid \forall g \in G\} \end{aligned}$$

where $\rightarrow$ indicates a directed edge in the graph.

Next, the $\vee$ or $\wedge$ nodes are added when we encounter files with such credentials. This is done during the preprocessing step while assigning files to their appropriate barrels (as described later in §4). For each such file, we insert a new vertex V_C for the file's credential C and adjust the graph to include new edges into and out of V_C as follows:

$$\begin{aligned} Dom(C) &= \{V_{C'} \mid C' \Rightarrow C\} \quad (II) \\ minDom(C) &= \{V \in Dom(C) \wedge \neg\exists V' \in Dom(C), V \in Dom(V')\} \\ Sub(C) &= \{V_{C'} \mid C \Rightarrow C'\} \\ maxSub(C) &= \{V \in Sub(C) \wedge \neg\exists V' \in Sub(C), V \in Sub(V')\} \\ E_{ACG} &= E_{ACG} \cup \{V \rightarrow V_C \mid \forall V \in minDom(C)\} \\ &\quad \cup \{V_C \rightarrow V \mid \forall V \in maxSub(C)\} \\ E_{ACG} &= E_{ACG} - \{V_1 \rightarrow V_2 \mid \forall\, V_1, V_2, V_1 \in minDom(C) \\ &\quad \wedge V_2 \in maxSub(C) \wedge V_1 \rightarrow C \in E_{ACG} \wedge \\ &\quad C \rightarrow V_2 \in E_{ACG}\} (removing\ redundant\ edges) \end{aligned}$$

This completes the construction of ACG. Using this, we can map files to appropriate barrels and also identify barrels searchable by a user (equivalent to finding V_u^* – simple depth first search on ACG).

3.2. ACB Minimality

Our ACB construction in §3.1.2 satisfies the following minimality property (Refer to [22] for proof).

Claim: *It is impossible to reduce number of ACBs without either duplicating files in barrels or violating ACAS.*

The total number of barrels in the system is equal to the number of distinct access control permissions in the system. While this number is usually linear in number of users and groups in real systems, theoretically this could be exponential. We have developed an optimization technique based on the ACB minimality, to address this rare problem. Our optimization transforms ACG with the goal of decreasing the number of ACBs. This transformation preserves *searchability*, that is, if a file f is accessible to user u, then in any transformed ACG, the file f belongs to some barrel b such that b is reachable from V_u on the ACG ($b \in V_u^*$). We call this *reachability* on the ACG.

Our technique aims to reduce the number of ACBs while satisfying ACAS at the cost of maintaining duplicates of files in indices. Let us consider any vertex V_C in the ACG such that credential $C \neq u$, for any user $u \in U$. One can eliminate V_C from ACG by adding all file indices in V_C to every vertex $v \in minDom(V_C)$, where $minDom$ is as defined in Equation-II. If V_C is reachable from some vertex V_u (for user u and $C \neq u$) then at least one vertex $v \in minDom(V_C)$ is reachable from V_u; thus the above construction satisfies reachability on ACG and preserves searchability. The construction preserves ACAS since the credential $domC$ associated with any vertex $v \in minDom(V_C)$ dominates the credential C ($domC \Rightarrow C$). Hence, any user u that satisfies the credential $domC$ also satisfies the credential C.

In our implementation we define a tunable parameter $minf$ – the minimum number of files per barrel (Our ACB construction in Section 3.1.1 achieves $minf = 1$). If a larger $minf$ is chosen the number of barrels decreases at the cost of more duplication of files indices. Given the parameter $minf$ we present a greedy algorithm to reduce the number of barrels as follows. (i) Sort the barrels in increasing order on their size (number of files) $b_0, b_1, \cdots b_k$. (ii) Pick the smallest i such that $b_i < minf$ and the credential associated with barrel b_i is not equal to u for any user $u \in U$. If there is no such barrel the procedure terminates. (iii) Eliminate the barrel b_i. Note that this may change the size of other barrels; so we resort the barrels according to their size and repeat the procedure.

It is also possible to design an optimization technique that reduces the number of ACBs while maintaining a single copy but allowing controlled violations of ACAS [22].

4. System Architecture

So far, we have introduced access control barrels (ACBs) and user access hierarchy as tools to (a) efficiently map files to ACBs and (b) determine accessible ACBs for a querying user. In this section, we explain the architecture of our indexing and search system.

Figure-2 shows the architecture. One enterprise machine serves as a global orchestrator and is responsible for managing the distributed environment. We will explain its various components in the next subsections. All participating machines run a thin client version of the system and are responsible for barrel indexing and query processing. Finally, the personal machines can integrate their local desktop search

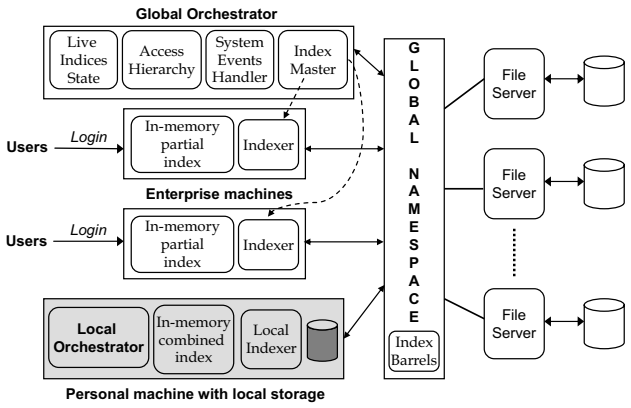

Figure 2: Distributed Indexing and Search

with enterprise search by running a local orchestrator agent, an added advantage of our approach.

4.1. Pre-processing: Creating ACBs

As part of the pre-processing step, we first create the basic ACG from the user and user groups in the system as described earlier. For *nix systems user and group information is obtained from */etc/passwd* and */etc/group*. Next, we initiate a filesystem traversal for all data that needs to be indexed. During the traversal, we associate each file with a vertex in the directed ACG graph based on its searchability privileges by finding the vertex that has the same credential as the file (e.g. V_{bob} for credential uid_{bob}). If the file has a $\vee/\wedge$ credential, a new node is added to the access hierarchy and the file is mapped to that node. At the end of this traversal, we have all barrels in the system and the list of files that are contained in each such barrel. These lists are written to per-barrel files, that are *securely* stored in the enterprise namespace with access only to the superuser. This stored file is the embodiment of our abstract ACB concept. This completes pre-processing and is usually performed by a single enterprise machine – the *global orchestrator*, which stores the user access hierarchy.

4.2. Indexing and Search

After creating ACBs, the next step is to index documents for each barrel. These ACBs can be indexed independently unlike the single index approach where the computation of TFIDF statistics requires centralized indexing of data. The *index master* in the global orchestrator distributes this barrel indexing task to participating enterprise machines. As barrels are stored in a global namespace and accessible to all enterprise machines, the orchestrator only needs to pass the barrel IDs to these machines. The orchestrator can easily optimize available resources by doing an intelligent distribution of barrels to machines. As we show later in §5, this indexing task distribution provides excellent savings.

On receiving commands from index master, the *indexer* component of enterprise machine agents retrieves barrels from the global namespace and starts indexing. The indices are then stored back into the global namespace. The access privileges based design of barrels provides a natural way of storing indices securely. The index files are stored with the same privileges as the files contained in that barrel. This allows only the users that had access to files of a barrel (and thus can search through them) to obtain these indices.

In our approach, querying and search can also be handled in a distributed fashion. When a user logs into an enterprise machine, the agent on that machine retrieves the indices that are accessible to that user, from the global namespace and caches them in memory. Now whenever a user queries these indices, search can be handled completely in a local environment, saving on (a) query response time and (b) resource requirements of a centralized search server. Note that all available enterprise search products today [4, 1, 12] have to use a highly capable search server (or a cluster) in order to deal with enterprise environments and querying always involves a network hop. In contrast, by integrating access control in a distributed fashion, we can reduce such requirements. Please note that in cases when the indices are too big to fit in the local cache, a central search server based design would be more suitable.

4.3. Handling Updates

In an enterprise environment, there are regular updates to file content and access privileges. This task is handled by the global orchestrator which subscribes to filesystem event notifications using available tools like inotify [17]. Once an event is received the orchestrator might need to make various kinds of changes. Change of file content is handled at a per-barrel basis by requiring that barrel indices to be appropriately modified (it does not require indexing the entire barrel again). This event is common to all enterprise search techniques and the ACB based approach does not incur additional overheads. In case of events when access permissions are modified that impact searchability, a document might need to be removed from one barrel and added to another (most indexers can handle this in an incremental manner as well), making it a low-cost event. Another event is the case of user/group membership modification, in which case the access hierarchy needs to be adjusted. A user/group addition is handled by adding a new node and corresponding edges (as done during initial ACG construction). Group membership modification is handled by changing the edges in the directed graph. Finally, a user/group deletion is handled by removing the appropriate node and all edges coming into or out of that node. All these operations are on in-memory ACG graph and are efficient. A detailed evaluation of this in comparison to the index-per-user approach is available in [22].

5. Evaluation

In this section, we present a detailed evaluation of our approach. All experiments were done on a Pentium-III Linux machine with 512 MB RAM and storage mounted via NFS. In these experiments we compare our ACB based approach with the current single-index enterprise search. Recall that the single-index approaches do not satisfy ACAS. We start with the datasets used in our experiments.

5.1 Datasets

The first data set, T14m, is a publicly available cleaned subcollection [14] of TREC Enterprise track (TREC 14) [7]. TREC 14 is a track for enterprise search and includes data from the World Wide Web Consortium (W3C) filesystems. The T14m dataset characteristics are shown in Table-1. It includes emails (*lists*), web pages (*www*), wiki web pages (*esw*) and people pages (*people*), but does not include any access control information.

Scope	Docs	Size	Avg. Doc Size
lists	173,146	485 MB	2.9 KB
www	45,975	1001 MB	23.8 KB
esw	19,605	80 MB	4.2 KB
people	1,016	3 MB	3.1 KB
Total	239,742	1569 MB	6.9 KB

Table 1: T14m: Cleaned TREC 14 subcollections

We also collected statistics from a real multiuser *nix enterprise installation, whose characteristics are shown in Table-2. We collected a subset of the entire directory structure with actual access control settings for 339,466 files arranged in 23,741 directories and replicated the structure in our test environment. The T14m data was used as content for the files (duplicating documents to fill all 339,466 files).

Number of users	926		
Number of user groups	1203		
Number of fi les	339,466		
Number of dirs	23,741		
Max depth of dir structure	23		
Size of data	2.05 GB		
Number of barrels	2132		
Barrels per user	Max	Avg	Median
	25	6.31	4.26
	21*	5.78*	3.96*

Table 2: Real enterprise dataset; Barrels/user was also computed at a second enterprise (shown by *)

5.2. Indexing Experiments

Indexing is a very important component of our approach. It includes a pre-processing step that creates the user access hierarchy and access control barrels followed by actual content indexing of the files contained in ACBs.

5.2.1 Pre-processing

As pre-processing performance is entirely dependent on the enterprise infrastructure (users/groups and directory structure), we use the real enterprise dataset. Figure-3 shows the evaluation of our implementation.

Task	Perf.
Hierarchy creation	38.7 sec
Barrel creation	263.1 sec
# Files `stat`'ed	202,446 (60%)
# Dirs `stat`'ed	14,059 (59%)

Figure 3: Pre-processing performance

Type	#Max-Docs	Time (s)
CSI	339,466	4640
BDI	189,546	2902

Figure 4: Indexing times; #Max-Docs are documents in the largest barrel

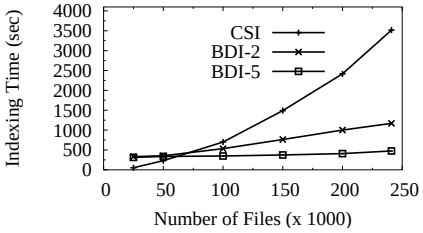

Figure 5: Indexing T14m dataset

Creating access hierarchy for 926 users and 1203 groups took 38.7 seconds, which is a small fraction of the total indexing time. It took 263.1 sec to create all ACBs. Additionally only 60% of the filesystem tree needed to be traversed to create barrels as in many cases, a higher level directory was mapped to a restrictive credential (e.g. only uid_{bob} can access) in which case its contents are automatically added without deeper traversal. Overall, these costs are only 10% of the distributed indexing approach (Figure-4).

5.2.2 Content Indexing

For indexing, we used the *arrow* indexing and search component of the Bow Toolkit [18]. We modified the ranking algorithm to the distributed algorithm of [16]. We consider two architectures: (1) *Centralized Single Index (CSI)* - representing current enterprise search products (that are *not* ACAS compliant), and (2) *Barrel-based Distributed Indexing (BDI)* - our barrels based distributed approach. *BDI-m* indicates that m machines are used to index barrels.

Figure-5 shows the time to index documents of the T14m dataset. For BDI approaches, documents were equally divided among participating machines. From the graph, CSI outperforms the BDI approaches when number of documents is small due to the pre-processing costs that BDI approaches incur. As the number of documents increases, BDI quickly outperforms CSI. The distribution of data into ACBs allows us to exploit available enterprise machines for faster indexing (an 85% improvement for 240K files).

The results for the T14m dataset above are a little optimistic as it considers a uniform size and distribution of barrels. However, in reality some barrels can be larger than others. To evaluate this, we performed indexing for our real

enterprise dataset. As shown in Figure-4, the barrel for `all` node (files that can be read by all users) was significantly larger and took longer than all other barrels combined (and thus total time does not vary with # machines). However, it was still 38% more efficient than CSI. In general, distribution is helpful when there are many large barrels and we expect that to be true in an enterprise environment.

5.3. Search Experiments

Recall that searching in our approach requires combining multiple barrels. However, for small number of barrels per user, overheads should not significantly deteriorate query performance. Secondly, since our approach does not require access control filtering at runtime, there would be savings as compared to the CSI approach.

For the querying experiments we used 150 queries obtained from TREC 14 Email search. The queries had an average of 5.35 terms per query. The results for CSI and n-BDI (where n is the number of barrels combined) are reported in Table-3.

Type	Index size	Loading time	Avg. time / query
CSI	230 MB	2.5 s	131.12 ms
2-BDI	258 MB	3.37 s	112.89 ms
5-BDI	269 MB	5.68 s	130.68 ms
10-BDI	280 MB	6.90 s	149.90 ms

Table 3: Search performance for T14m *lists*; Loading time is the time to load all indices in memory

Notice that the BDI approaches have slightly larger indices since they have to store many words multiple times in different barrel vocabularies. Next, the time to load indices into memory also increases with the number of barrels as there are more file I/Os to gather the index data. However, this is only a one-time cost and once indices are cached, queries proceed normally. Finally, the average query time for BDI approaches is comparable to CSI with 2-BDI and 5-BDI even outperforming it by saving on the privileges check required at runtime in the CSI approach.

We also compared the ranking of the BDI approaches to CSI ranking. For this we evaluated the percentage of top-10 results of the CSI approach that occurred in top-100 of the distributed approach and their average ranks. As shown in Table-4, for our average case of 5 barrels per user, nearly 70% of top-10 results occurred in top-100 of the BDI approach with an average rank of 14. We believe that ranking can be further improved using more sophisticated distributed ranking [23, 19].

Type	10-in-100	Avg. Rank
2-BDI	75%	13
5-BDI	68%	14
10-BDI	61%	15

Table 4: Ranking comparison for TREC 14 *lists*. 10-in-100 is the % age of CSI top-10 results in top-100 of x-BDI and avg-rank is the average rank of CSI top-10 results in x-BDI top-100

6 Conclusions

In this work, we presented an efficient and secure approach to enterprise search. We demonstrated the inadequacy of existing solutions at ensuring access control aware search and developed distributed techniques that elegantly capture access control semantics of enterprise filesystems, using *access control barrel (ACB)* and *user access hierarchy* concepts. The distributed and parallel nature of our solution helps improve indexing efficiency and reduces resource requirements for search servers. We also described an optimization that improve the scalability of our approach even in complex settings. Our experimental evaluation on real datasets shows improved indexing efficiency and minimal overheads for ACB processing.

References

[1] Coveo enterprise search. *http://www.coveo.com*.
[2] Gartner group. *http://www.gartner.com*.
[3] Google desktop. *http://desktop.google.com*.
[4] Google enterprise. *http://www.google.com/enterprise*.
[5] Linux Manual Pages. *man command-name*.
[6] MSN toolbar. *http://toolbar.msn.com*.
[7] TREC Enterprise. *http://www.ins.cwi.nl/projects/trec-ent*.
[8] Windows Desktop Search for Enterprise. *http://www.microsoft.com/windows/desktopsearch*.
[9] Yahoo desktop. *http://desktop.yahoo.com*.
[10] Butler Group. Unlocking value from text-based information. *Review Journal Article*, March 2003.
[11] M. Bawa, R. Bayardo, and R. Agarwal. Privacy preserving indexing of documents on the network. In *VLDB*, 2003.
[12] S. Büttcher and C. Clarke. A security model for full-text file system search in multi-user environments. In *FAST*, 2005.
[13] A. Grunbacher and A. Nuremberg. POSIX Access Control Lists. *http://www.suse.de/%7Eagruen/acl/linux-acls/online*.
[14] D. He. Cleaned W3C Subcollections. *http://www.sis.pitt.edu/%7Edaqing/w3c-cleaned.html*.
[15] M. Kallahalla, E. Riedel, and et al. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.
[16] O. Kretser, A. Moffat, T. Shimmin, and J. Zobel. Methodologies for distributed information retrieval. In *ICDCS*, 1998.
[17] R. Love and J. McCutchan. inotify linux file system monitor.
[18] A. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. *http://www.cs.cmu.edu/%7Emccallum/bow*.
[19] A. Powell, J.French, J. Callan, and et al. The impact of database selection on distributed searching. *SIGIR*, 2000.
[20] D. Ritchie and K. Thompson. The UNIX Time Sharing System. *Communications of the ACM*, 17(7), 1974.
[21] S. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC-7. In *TREC*, 1998.
[22] A. Singh, M. Srivatsa, and L. Liu. Efficient and Secure Search of Enterprise File Systems. *Georgia Tech CERCS Technical Report GIT-CERCS-07-07*, 2007.
[23] J. Xu and J. Callan. Effective retrieval with distributed collections. In *SIGIR*, 1998.